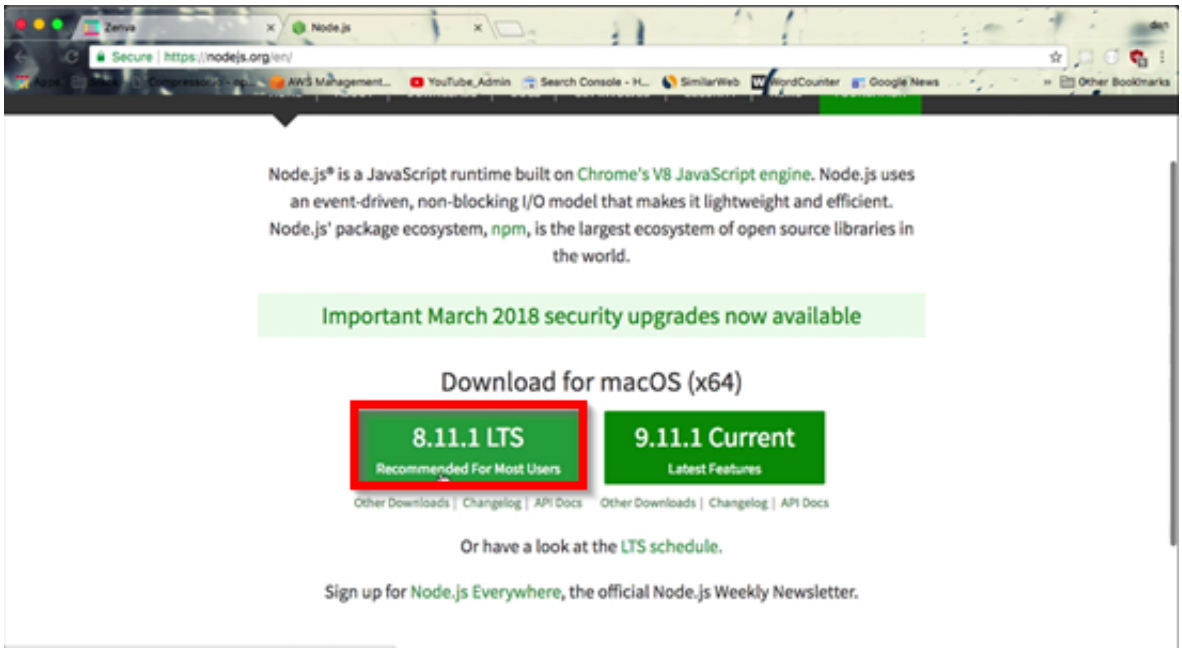
Welcome to **Zenva's Express for Beginners** course. In this course, we'll be covering the **Express framework** (https://expressjs.com/) used for web applications in **node.js**. For this lesson, we'll be installing necessary tools and getting our basic environment set up.

Setting up Node.js

To begin, we need to make sure to **download** and **install** the latest **LTS** version of **node.js** (https://nodejs.org/en/). This will be the most stable version of **node.js** that allows us to simulate a server to test our application.



Once you have **node.js** installed, open **Terminal** on **Mac** or **Command Prompt** on **Windows**. By typing in the following, you can verify that you have the most current version of **node.js**:

```
node -v
```

Now, using **Terminal/Command Prompt**, navigate to your desktop.

```
cd ~/desktop
```

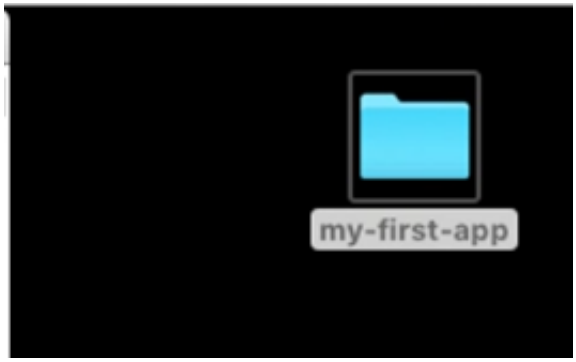
If using **Windows**, the following code may be more appropriate if you are in your profile directory:

```
cd desktop
```

Next, we'll create a **new directory** on the desktop called **my-first-app**.

```
mkdir my-first-app
```

On your desktop, you should now see a new folder to contain our project ready and waiting for you. We will continue working in **Terminal/Command Prompt**, but you can create folders and files via your operating system GUI.



Back in **Terminal/Command Prompt**, **change** into that project **directory**.

```
cd my-first-app
```

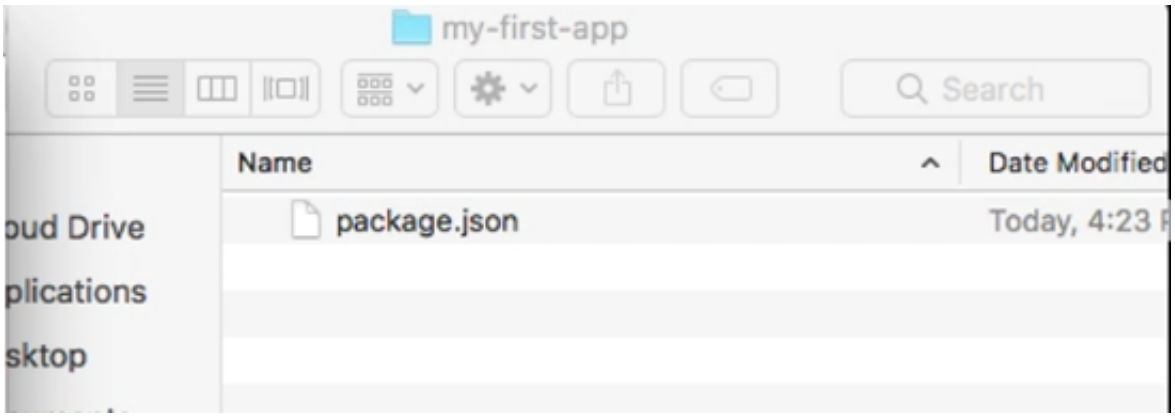
We can now create our **node project** by running the command below. This will add in the necessary **package JSON** to serve as a configuration file for the project.

```
npm init
```

During this process, you will be prompted about things regarding the file. For most of the entries, you can simply hit **enter** and accept the defaults. When it comes to **description**, though, we'll enter **"my first express project"**. Before confirming **yes**, verify that the file being written appears like below:

```
{
    "name": "my-first-app",
    "version": "1.0.0",
    "description": "my first express project",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "",
    "license": "ISC"
}
```

You can verify that the file was created by checking the **my-first-app folder** and locating the **package.json** file.

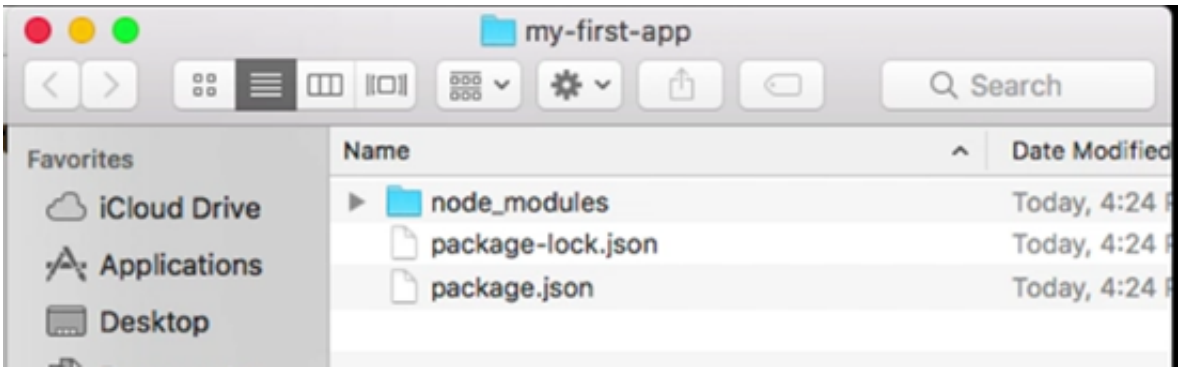


Installing the Express Framework

Our next task is to install the **Express framework**. Using the following command, make sure to use **–save**, as this will automatically add **required dependencies** to the **package.json** file for you.

```
npm install express --save
```

Once again, you can check the **my-first-app folder** to verify all the necessary components were added. Besides **package.json**, you should also have **package-lock.json** and a **node_modules folder**.



The next step in our setup is to create our **entry point**, which will be **index.js** as was specified in our **package.json file**. Make sure to use the command below that is appropriate for your **operating system**. If using Windows, also make sure to delete the placeholder text that appears in the new file.

Mac:

```
touch index.js
```

Windows:

```
echo index > index.js
```


Once again, you should see the new file within the **my-first-app folder**.

First Get Handler

For this course, we'll be using the **Sublime text editor** (https://www.sublimetext.com/) to write our code. If you don't have a preferred code editor, please be sure to **download** and **install Sublime** before we begin.

With **Sublime** (or other code editor), **open** up the **my-first-app folder** so we can begin working on the project. In our **index.js** file, the first thing we want to do is **import** our **Express framework** and also get a **reference** to our **Express application** itself.

```
const express = require('express')

const app = express()
```

Our next step is to define a **route** for **incoming requests**. Since **get requests** are the most common kind servers receive, we'll be creating what's known as a **get route handler**. Within this **handler**, we need to pass in **two arguments**. The first **argument** is the **route** itself, whether the home page or some other page that is part of the application/URL. The second **argument** is a **function** that takes **three arguments**. We will worry about the third **argument** in a bit, but know that the first **argument** in the **function** is the incoming request and the second **argument** is the **response** we want to occur when that **request** comes through.

With our first **route handler**, we'll be using a **backslash** to designate the **home page** as our **route/path**. Thus, anytime the user tries to access the home page, they will be serviced by this specific **handler**. For the **function**,we will use the built-in **send function** to send back a simple **string** as our **response**.

```
app.get('/', (req, res, next) => {
    res.send('My First Express App!')
})
```

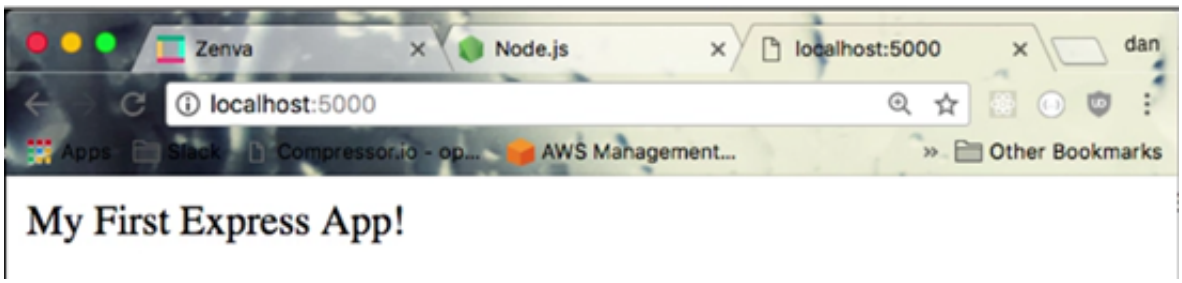
Let's test our **handler**. To do this, we need to add the following line at the bottom of **index.js**. With this code, we assign a **port number** (in this case **5000**) so that we can open a **local host server** that only exists on our computer. While an advanced topic, just know that browsers can be set to listen to different ports for different functionality.

```
app.listen(5000)
```

In **Terminal/Command Prompt**, we need to set up the server using the **index.js** file to run it.

```
node index.js
```

Finally, in your browser, you can open **localhost:5000** as the address and see the response.



If you'd like confirmation in **Terminal/Command Prompt** that the server is running, you can use a **console log** command to notify you. This should go below your **app.listen** command in **index.js**.

```
console.log('Server running on http://localhost:5000')
```

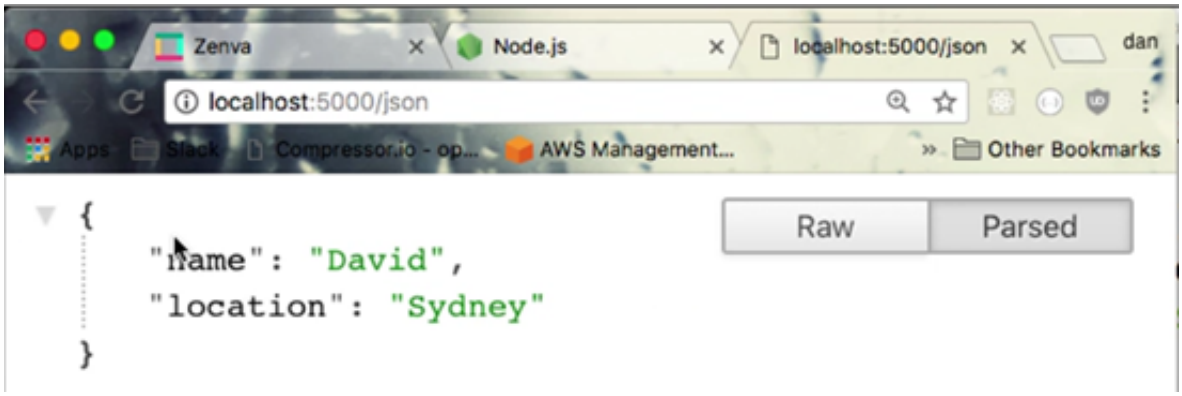
Each time you want to test your page, hitting **Ctrl/Command + C** will terminate the server. Then, you can start the server again to reload all the information This will be a necessary step throughout the course so keep it in mind.

Other Response Types

Now that we can create a basic **route handler**, let's try two more that have different **response types**. This time, we'll create a **route handler** that activates when we try to go to **localhost:5000/json**. In regards to our **response**, instead of **send**, we will use **json** for this one. This **response type** will return a standard **Javascript object**. To demonstrate, we'll create a **Javascript object** in the **handler's function** to return the appropriate **response type**.

```
app.get('/json', (req, res, next) => {
 const data = {name:'David', location:'Sydney'}
 res.json(data)
})
```

Make sure to shutdown and start up your server in **Terminal/Command Prompt**. Then, in your browser, open **localhost:5000/json**. You should now see the **Javascript object** as the response.



Our next test case will be to respond back with formatted **HTML**. This time, we will use a **send function** to send back the response. However, within the **string** we send back, we'll use various **HTML markup**.

```
app.get('/html', (req, res, next) => {
 const html = '<html><h1 style="color:red">This is an HTML response</h1></html>'
 res.send(html)
})
```

Restarting your server again, you can try **localhost:5000/html** (the **route** we set up above) to see the page in action. As you can see, the **HTML** renders as HTML instead of plain text.

In the next lesson, we're going to work with more dynamic **responses** and paths using **query strings** and **route parameters**.

You can learn more about routing with **Express** by checking out the documentation: https://expressjs.com/en/guide/routing.html

If you are using Windows, you can not use the “touch” command to create files. However, you can do the same using the “echo” command from Windows.

So, instead of using

```
touch file_name
```

you can use

```
echo file_name
```

In this lesson, we're going to set up more **dynamic route handlers** that utilize **query strings** or **route parameters** to send a **response**.

Query String Routing

Back in **index.js**, we will now add a new **get route handler** underneath the ones we created last lesson. This time, our **route** we'll be **/query**. To get our **query's** contents, we need to **extract** them from the route in the URL. Thankfully, there is a **built in function** called **query** that can automatically do this for us based on the **request** sent in. For this **handler**, we will simply return the **query** string in a **JSON** format using the **.json** function we used last lesson.

```
app.get('/query', (req, res, next) => {
 const query = req.query
 res.json(query)
})
```

Now we need to restart the server. As you might have noticed, this is getting tedious. We can be more efficient about this by **globally installing nodemon**. **Open Terminal/Command Prompt** and enter the following command:

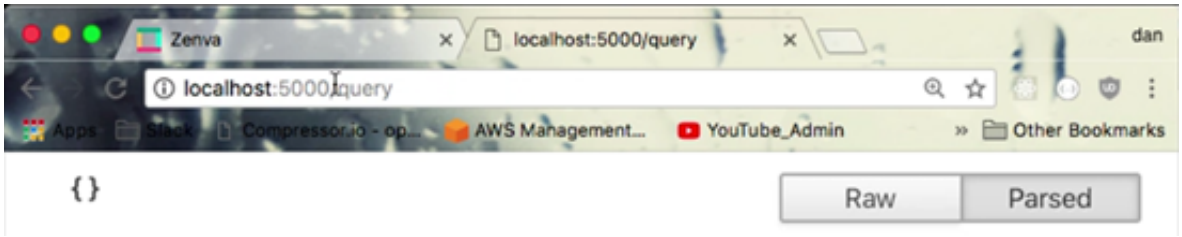
```
sudo npm install nodemon -g
```

Making sure you're in the **directory** for your **project** (i.e. the **my-first-app folder**), run **nodemon** so that it can work.

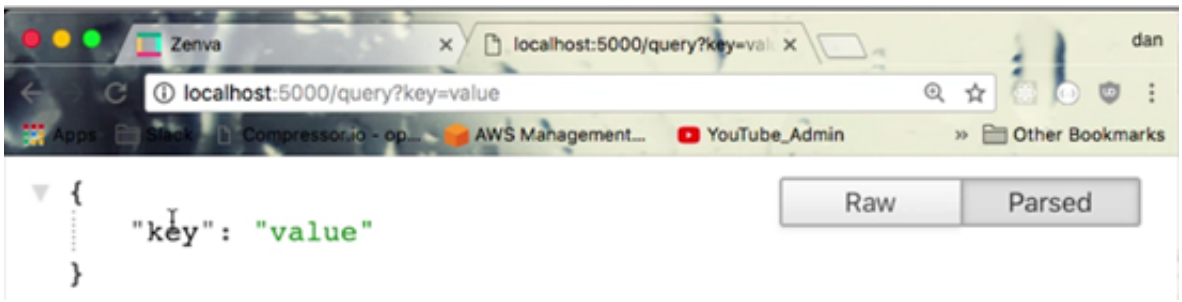
```
nodemon
```

With **nodemon**, our files will be constantly watched. If anything changes in those files, it will automatically relaunch the server for us! Thus, we just need to leave the server running in the background without us needing to manually restart it so much.

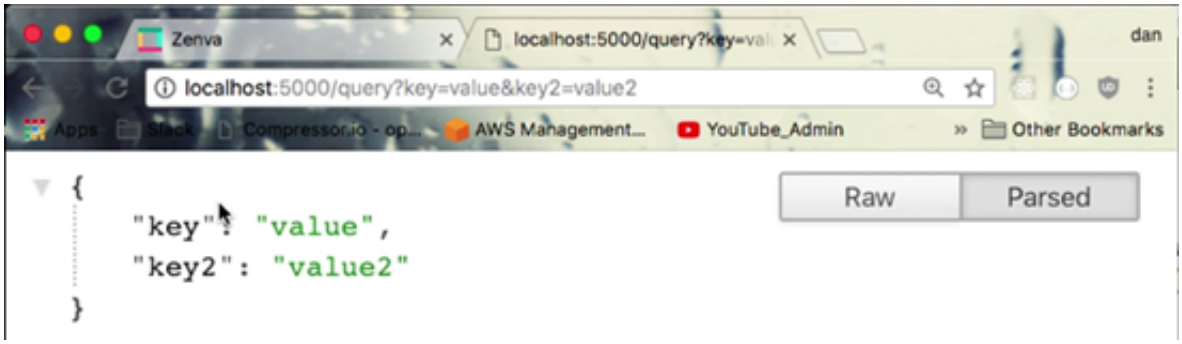
With that done, we can open up **localhost:5000/query** to test our query. Initially, you should receive an empty, raw **Javascript object**.



This is because we need to **append** a **query string** to the **URL** with a **question mark** in order for us to see any content. For **query strings**, they are always divided into a **key** and **value**. If you try **localhost:5000/query?key=value**, you should see a fuller object.



Using an **ampersand**, you can also add even more **key value pairs** to the **query**. As such, our **query strings** can be as long as we want.



Thanks to the **Express framework**, our URL is automatically parsed for the string without us needing to do extra work. Thus, our response is more **dynamic** and isn't required to be **static** or **fixed** like our previous **route handlers**.

Route Parameter Requests

Let's move on to creating a **route handler** that utilizes **route parameters**. In order to use the **route parameters**, we need to note the **parameters** in the route itself using a **colon**.

```
app.get('/params/:name/:location/:occupation', (req, res, next) => {

})
```

Like with our **query string**, the **Express framework** provides us with a built-in **function** called **params** that can automatically extract our **parameters** from the **route**. With our **parameters** extracted, we can once again create a **JSON object** that will **respond** with the **parameters**.

```
app.get('/params/:name/:location/:occupation', (req, res, next) => {
 const params = req.params
 res.json({
  params: params
 })
})
```

Unlike **query**, when we go to test this in our browser, we need to fill out all three parameters. In other words, we can't just have a URL with the name and occupation parameters or any mix. The

URL must contain all three to work (or whatever number of parameters you declare).

Let's quickly test our new **route handler** by going to **localhost:5000/params/mike/brisbane/doctor**.

As you can see, all three values get returned back as a **Javascript object** in **JSON form**.

Summary

With **queries** and **route parameters**, we can create **dynamic responses** that don't require a fixed entry point. However, while **query strings** can be endless, **route parameters** must strictly adhere to the format defined in the **route**.

In the next lesson, we'll work on our project's structure and make it more typical to what you would see in an application.

Remember that if you'd like to learn more about routing, you can check the **Express documentation:** https://expressjs.com/en/guide/routing.html

In this lesson, we're going to work on creating a project structure that you might see used in a sophisticated web application.

Introduction and Basic Setup

While you can organize a project in any fashion, there are some typical patterns seen across various web applications. By **Googling "node express project structure"**, you can check out some of the ways that are popular among developers. There are also tools that will automatically scaffold our project for us, which we will be taking a look at today.

In **Terminal/Command Prompt**, navigate back to your **desktop**.

Mac:

```
cd ~/desktop
```

Windows:

```
cd desktop
```

Next, we will use the following command to **globally install** a generator that will scaffold a project for us.

```
sudo npm install -g express-generator
```

With this installed, any new projects will automatically be structured in a typical, usable way. With the following command, we will create a new project folder called **project** to view the structure. This is optional and not something you have to do. We're just going to look at the folder setup.

```
express project
```

We will open this new **project** in **Sublime** to view the structure. Automatically, we should see a few folders in the **project folder**.

- **Bin**, which we won't cover right now
- **Public**, where all the static assets go
- **Routes**, where the routes are organized
- **Views**, where all our automatically generated HTML templates go

This is a pretty simple but standard structure, so we're going to implement it in our **my-first-app project**.

Structuring our App: Routes

Back in **Terminal/Command Prompt**, navigate back to the **my-first-app folder**. You should also **open** the **my-first-app folder** in **Sublime** if you followed along in the above section.

```
cd my-first-app
```

The first thing we'll do is create a **new directory** for **routes**. Then, we'll **change directories** to this new folder.

```
mkdir routes
```

```
cd routes
```

Next, we'll create a **new file** to contain our **routes**. Be sure to use the code appropriate for your operating system.

Mac:

```
touch index.js
```

Windows:

```
echo index > index.js
```

You should see the new file now in **Sublime**.

We need to copy the **route handlers** we made earlier into this **new index.js file** in the **routes folder**. Before that, though, we need to **import** both **Express** and a **router** so our **routes** can be utilized properly. Additionally, since we'll now be using this file from our main **index.js** file, we need to **export** this router so other files can use it.

```
const express = require('express')
const router = express.Router()

module.exports = router
```

Between the **imports** and **export**, we'll add in a test **route handler** for the **get request**. This is the same format as previous lessons, so nothing special is needed there. For now, we'll use a **path** for our **home page** and send back some simple text.

```
router.get('/', (req, res, next) => {
    res.send('This is the index route!')
})
```

Locate the **index.js** that's in the **main directory** and open it. At the top, we need to **import** our new **routes/index.js file** so it can be used.

```
const routes = require('./routes/index')
```

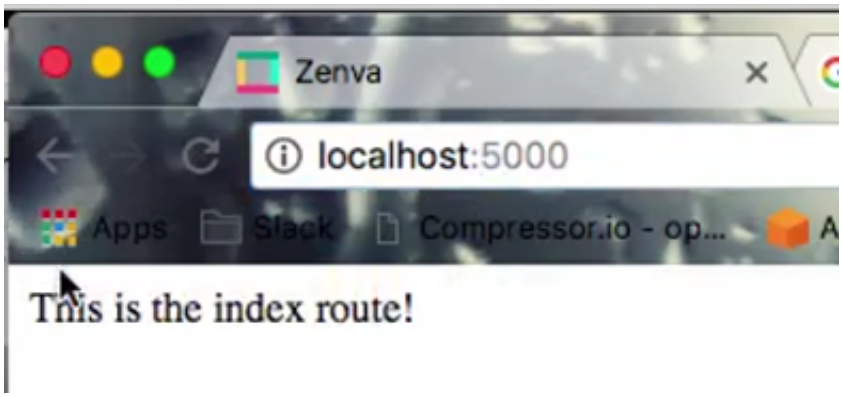
For now, **comment out** our previous **router handlers** with **//** so they don't interfere with our test. Also, we need to let our app know it needs to use our **router** file for **routes**. We can do this using **.use** and specifying the **path** and appropriate file (which we saved as a variable).

```
const express = require('express')
const routes = require('./routes/index')
```



```
const app = express()

app.use('/', routes) //The new line

//Commented out handlers from previous lessons

app.listen(5000)
console.log('Server running on http://localhost:5000')
```

You can test the project with **nodemon** very quickly to confirm that your setup is correct.



Now that we've confirmed the file is working, **cut** the **commented out get handlers** from the main **index.js** file and **paste** them into the **routes/index.js file**. You should overwrite our test **handler** from before. Additionally, make sure to change the object from **app.get** to **router.get**.

```
router.get('/', (req, res, next) => {
 const data = {
  name: 'Index',
  date: 'April 22, 2018'
 }

 res.render('index', data)
})

router.get('/json', (req, res, next) => {
 const data = {name:'David', location:'Sydney'}
 res.json(data)
})

router.get('/html', (req, res, next) => {
 const html = '<html><h1 style="color:red">This is an HTML response</h1></html>'
 res.send(html)
})

router.get('/query', (req, res, next) => {
 const query = req.query
 res.json(query)
})
```



```
router.get('/params/:name/:location/:occupation', (req, res, next) => {
 const params = req.params
 res.json({
  params: params
 })

})
```

Our program is now becoming more **modular**, which is the ideal setup for an application. While our **routes/index.js file** can handle our routes, our main **index.js** can now be responsible for **importing** all the necessary mechanics and setting up **configurations** to make our app function. Make sure to read the **Express documentation** for more information on routes and routers: https://expressjs.com/en/guide/routing.html

Structuring our App: Views

With **routes** handled, let's make a **new directory** for **views** via **Terminal/Command Prompt**. Then, after, we'll **change** into that **directory**.

```
mkdir views
```

```
cd views
```

(**Note:** Make sure to change into the **my-first-app** directory so you aren't in the **routes directory**).

As mentioned, the **views directory** generally stores **templates** used to render our pages. To use these, we use a **templating engine** which is able to **inject data** into a page, thus creating a dynamic output. While there are many **templating engines**, we're going to use the **Hogan templating engine (.hjs)**. You can learn more about this engine via its website and documentation: https://github.com/twitter/hogan.js

Head back into the **my-first-app directory** via **Terminal/Command Prompt** and **import** the **module** for it.

```
cd my-first-app
```

```
npm install hjs --save
```

In **Sublime**, **open** the **main directory's index.js file**. We need to tell this file about our **engine** so it can use it. Using the **.set function**, we can do this by specifying our **view engine** is **hjs**. We will add this under where we **use** the **routes**. However, we also need to specify that the **views directory** we created is the directory we need to use for our **templates**. To do this, we need to both **import** the **path module** to our project as well as **set** the app to know the proper directory we want for our **views**.


```
const express = require('express')
const routes = require('./routes/index')
const path = require('path') //Imports the path module

const app = express()

app.use('/', routes)
app.set('views', path.join(__dirname, 'views')) //Tells the code that are views direc
tory is to be used for rendering templates
app.set('view engine', 'hjs') //Sets up the view engine


app.listen(5000)
console.log('Server running on http://localhost:5000')
```

Now we can create our first template! Once again in **Terminal/Command Prompt**, navigate to the **views directory**. Then, using the command appropriate to your operating system, we'll create our file.

```
cd views
```

Mac:

```
touch index.hjs
```

Windows:

```
echo index > index.hjs
```

Open the new **index.hjs** file in **Sublime**. We will create a basic **HTML** page in the file for now.

```
<html>
<head></head>

<body>
 <h1>This is the Index Template</h1>

</body>

</html>
```

We have our **template**, but we need to tell our app when to **render** it. **Open** the **routes/index.js** file again and locate the **route handler** for the home page. Instead of our plain text, we'll have it render our **index.hjs template**. To do this, we have another **built-in function** called **render**. In this function, we first pass in the **name** of the **template** and the data

we're injecting (in this case **null** for none).

```
router.get('/', (req, res, next) => {
 res.render('index', data)
})
```

Running the server again, you can go to the home page of **localhost:5000** and see our **template** contents are indeed rendered.

Completing our Directory Structure

With **views** done, we're going to set up the last of our project structure. The last directory we need to create is our **public directory**. This directory, as mentioned, will hold the **static assets** for our project. This includes static pages, images, media files, and more.

In **Terminal/Command Prompt**, we will create this **public directory** and then **change** to it. Within that **directory**, we also want to make **three** more **directories**: one for our **CSS files**, one for our **Javascript files**, and one for our **images** assets.

```
mkdir public
```

```
cd public
```

```
mkdir css
```

```
mkdir images
```

```
mkdir js
```

(**Note:** Once again, make sure you're in your **my-first-app directory** before running the commands).

Our project is now properly structured.

Testing the Capabilities of Template Rendering

Before we close this lesson out, we mentioned earlier that we can **inject data** into our templates with the **view engine**. We haven't used that yet, so let's test it out. **Open index.hjs** (the template). Instead of our previous headline, we're going to create content that uses placeholders. With these placeholders, we essentially make a call to a **variable** whose data is determined by the **route handler**.

```
<html>
<head></head>

<body>
 <h1>This is the {{name}} Template</h1>
 <p>Today is {{date}}. My name is {{firstName}}.</p>

</body>

</html>
```

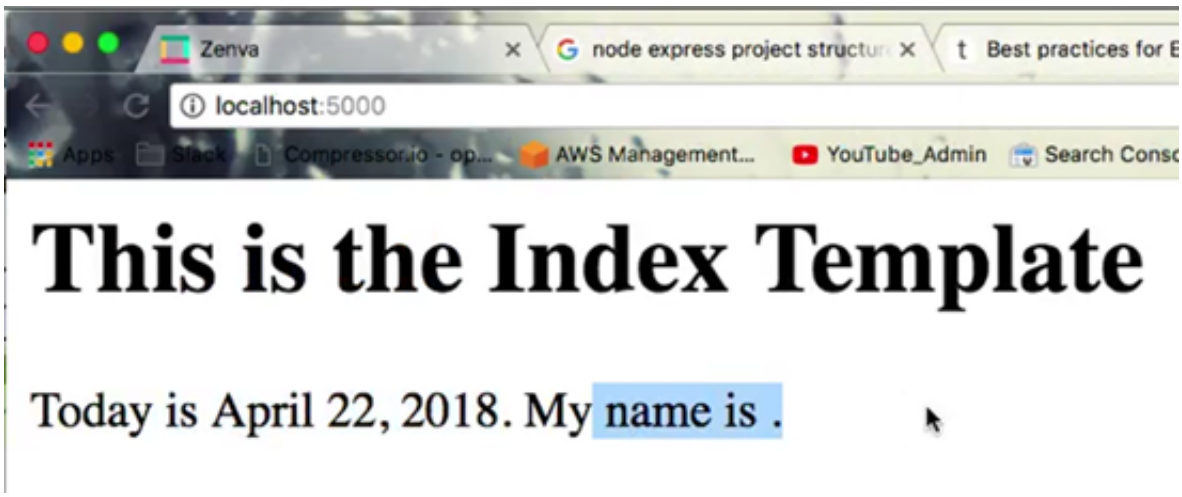
In **routes/index.js**, we can change our **home page route handler** to contain data. Making sure to use the same **variables** mentioned in our **index.hjs template**, we can set up a name and date for the page to use when it renders.

```
router.get('/', (req, res, next) => {
 const data = {
```



```
  name: 'Index',
  date: 'April 22, 2018'
 }

 res.render('index', data)
})
```

Running **nodemon**, we can now see this in action.



Notice that because we didn't set up a **key value pair** for the **firstName** placeholder, it renders as **empty**. However, with the others, our **data** is **injected perfectly**.

In the next lesson, we'll work with using static assets.

In this lesson, we're going to set up and connect our **static assets** in our **public directory** so we can use those files. If not already there, make sure to navigate to the **my-first-app directory** in **Terminal/Command Prompt**.

Adding a CSS File

The first thing want to do is connect up the **CSS directory** so we can render our pages using styles. In the **main directory's index.js file**, which contains our imports and configurations, we need to tell our app to use the **public directory** anytime a **static** asset is requested.

```
app.use(express.static(path.join(__dirname, 'public')))
```

Let's test this out by creating a **style sheet**. Navigate to the **css directory** in the **public directory** so we can create this new file. Once again, make sure to use the command appropriate to your operating system for creating the file.

```
cd public
```

```
cd css
```

Mac:

```
touch style.css
```

Windows:

```
echo style > style.css
```

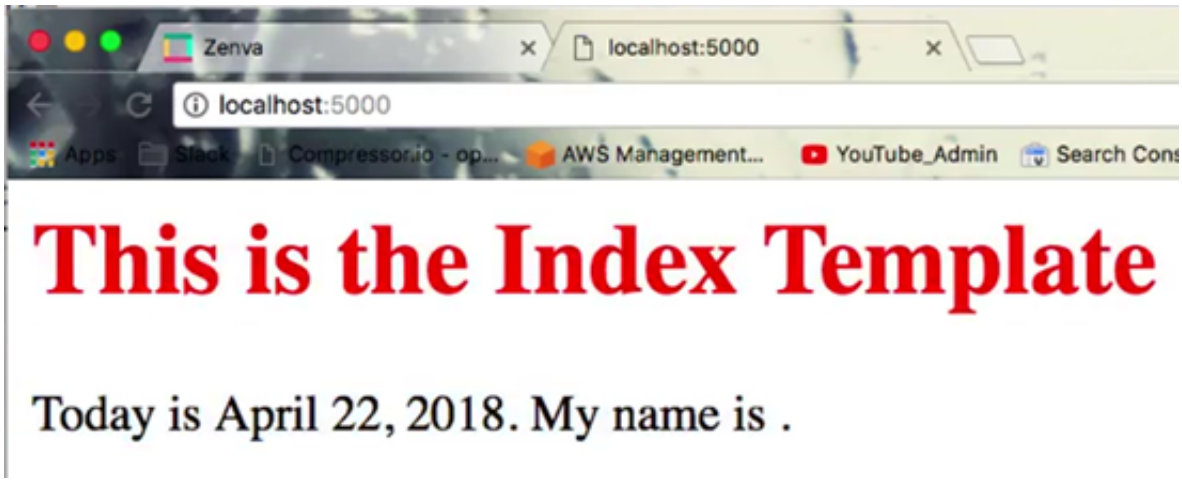
In **Sublime**, open this **style.css file**. For this file, we just want to simply change the **headline** to **red**.

```
h1 {
 color: red;
}
```

We need to connect this style sheet up to our **template** though. As such, **open** the **index.hjs file** and add the **import link reference** to our **style sheet** in our **head tags**. Notice that we don't specify the **public** folder in the **href** portion, as our app has already been told to check for **static assets** in the **public directory**.

```
<link rel="stylesheet" type="text/css" href="/css/style.css">
```


From the project root, you can run **nodemon** to test and see the results.



Importing Bootstrap

We can also **import** and use things like **Bootstrap**. In **Terminal/Command Prompt**, once again navigate to the **css directory** and create a new file.

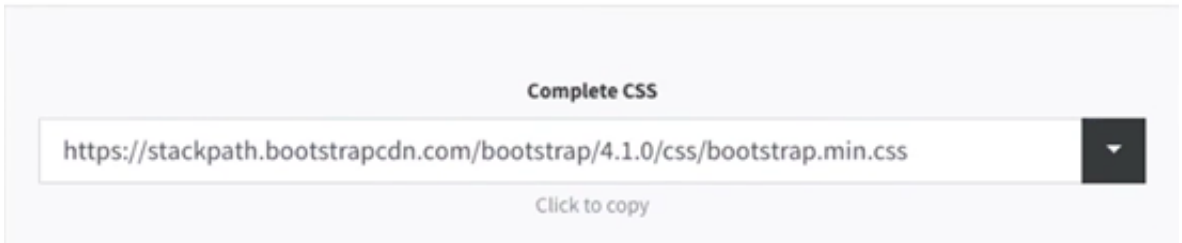
Mac:

```
touch bootstrap.min.css
```

Windows

```
echo bootstrap > bootstrap.min.css
```

Next, from the **Bootstrap CDN website** (https://www.bootstrapcdn.com/), we can **copy** and **paste** the **Complete CSS** path into a browser.



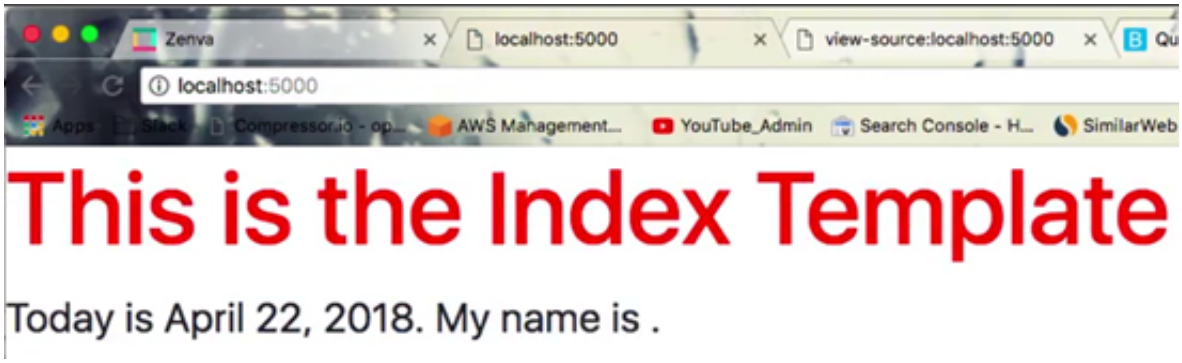
In this path, you should see the entirety of the **Bootstrap framework**. **Copy** this entire code and **paste** it into the new **bootstrap.min.css file** in **Sublime**. Now, in the **index.hjs file**, we can add another **import reference** for this **Bootstrap file**. Additionally, we'll use a **div container** to add more padding to all our content.

```
<html>
<head>
 <link rel="stylesheet" type="text/css" href="/css/bootstrap.min.css">
 <link rel="stylesheet" type="text/css" href="/css/style.css">
```



```
</head>

<body>
 <div class="container">
  <h1>This is the {{name}} Template</h1>
  <p>Today is {{date}}. My name is {{firstName}}.</p>
 </div>

</body>

</html>
```

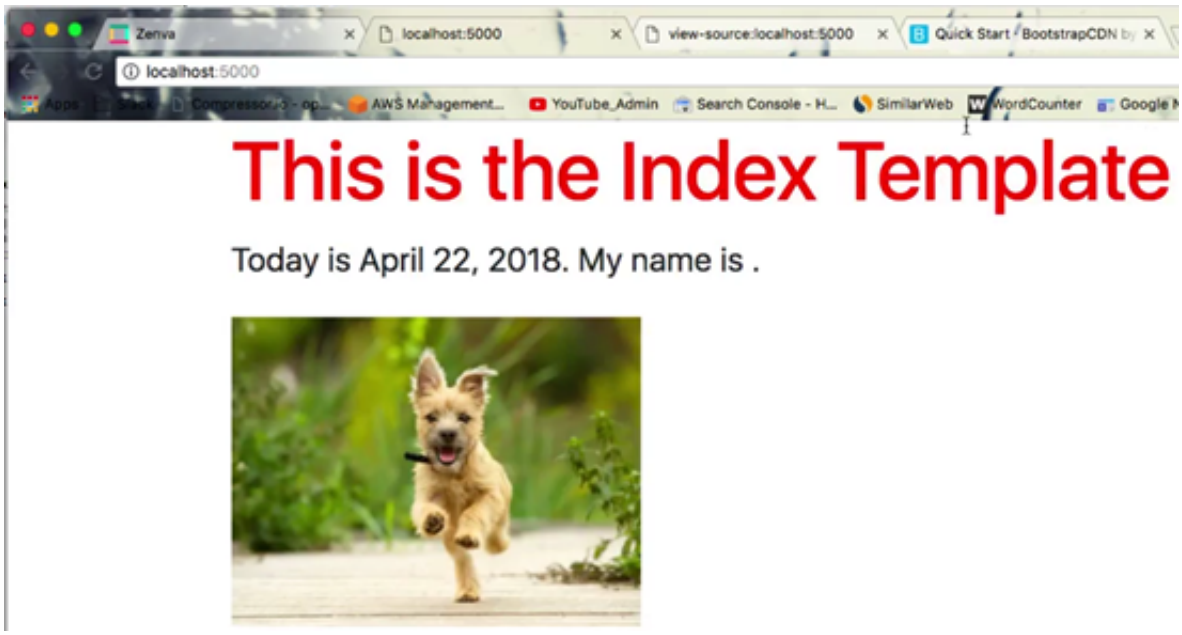
If you test the app once again, you should notice a slight change since the **template** is now using default **Bootstrap** styles.



Using an Image

The procedure for using any sort of **static asset** is going to be the same as above. However, just to demonstrate that fact, we will add an image to our **template**. Find any image you want (or locate the one we used in the **Course Files** for this lesson) and save it to your **public/images directory**. Then, in **index.hjs** under the **paragraph text**, we will add the **image tag** with a reference to our image. So the image isn't overly large, we will style the **width**. Noticeably, it still works as intended when we test the app.

```
<img style="width:200px" src="/images/puppy.jpg">
```

In the next lesson, we will return to our **templates** and learn how to do **data injection** with **loops** and **conditionals**.

If you'd like to learn more about using **static assets** in the **Express framework**, please check out the appropriate documentation: https://expressjs.com/en/starter/static-files.html

In this lesson, we're going to return to our **templating engine** and learn how we might **render arrays** and **conditionals**.

Dealing with Arrays

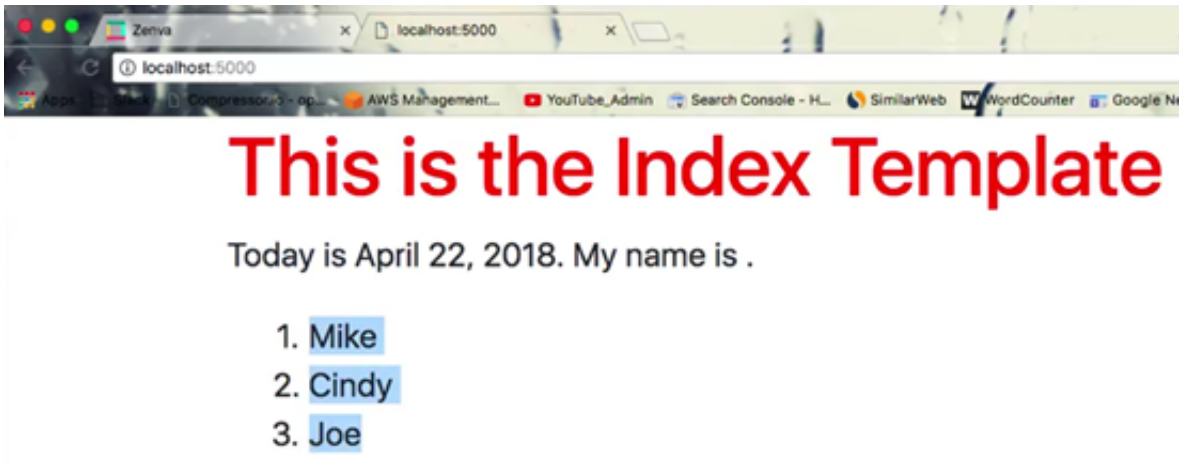
Instead of rendering our image from the last lesson, we're going to replace this with a **list** of profiles that is taken from an **array**. First, in the **routes/index.js** file, we're going to add our **data** to the **routes**. This time, though, our **value** will be an **array** of **objects**.

```
router.get('/', (req, res, next) => {
 const data = {
  name: 'Index',
  date: 'April 22, 2018',
  profiles: [
   {name: 'Mike'},
            {name: 'Cindy'},
            {name: 'Joe'}
  ]
 }

 res.render('index', data)
})
```

Open **index.hjs** now. In order to render this list in the **Hogan templating engine**, we need to use a **hashtag** and **forward slash** to denote that we want to use a **for loop** to **iterate** through all the items in our **array**. Within this loop, we are able to see all our **array's objects** and access the **key value pairs**, like **name**, to render those.

```
<ol>
  {{#profiles}}
    <li>
      {{name}}
    </li>
  {{/profiles}}
</ol>
```

You can test the app to see that the code does indeed render our data.

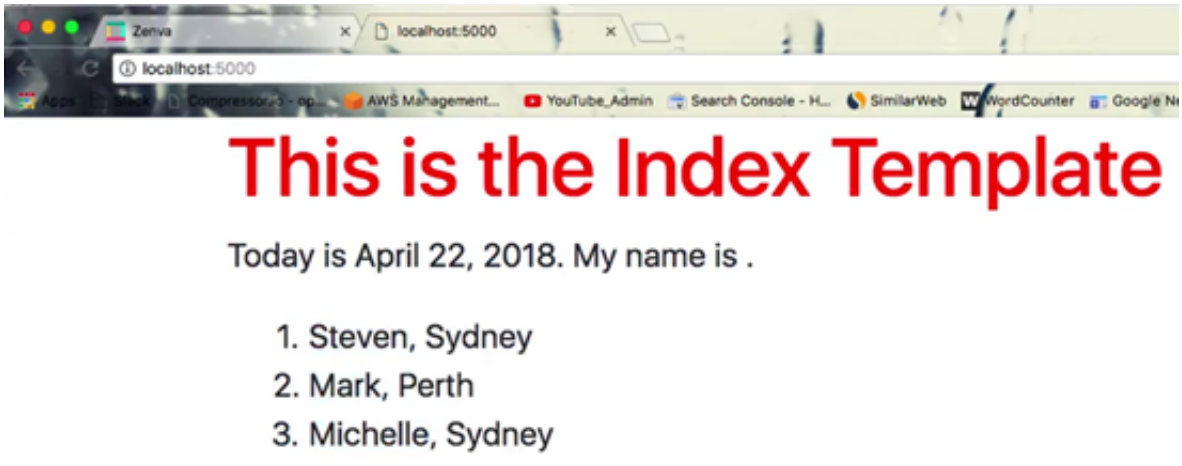


Given that our **array** is made of **JSON objects**, we can also add more **key value pairs** to the objects. In **routes/index.js**, we will add the following.

```
profiles: [
 {name: 'Steven', city:'Sydney'},
 {name: 'Mark', city:'Perth'},
 {name: 'Michelle', city:'Sydney'}
]
```

With a slight tweak to **index.hjs**, we can easily add the city **key** to our loop and have that printed as well.

```
<ol>
  {{#profiles}}
 <li>
   {{name}}, {{city}}
 </li>
  {{/profiles}}
</ol>
```



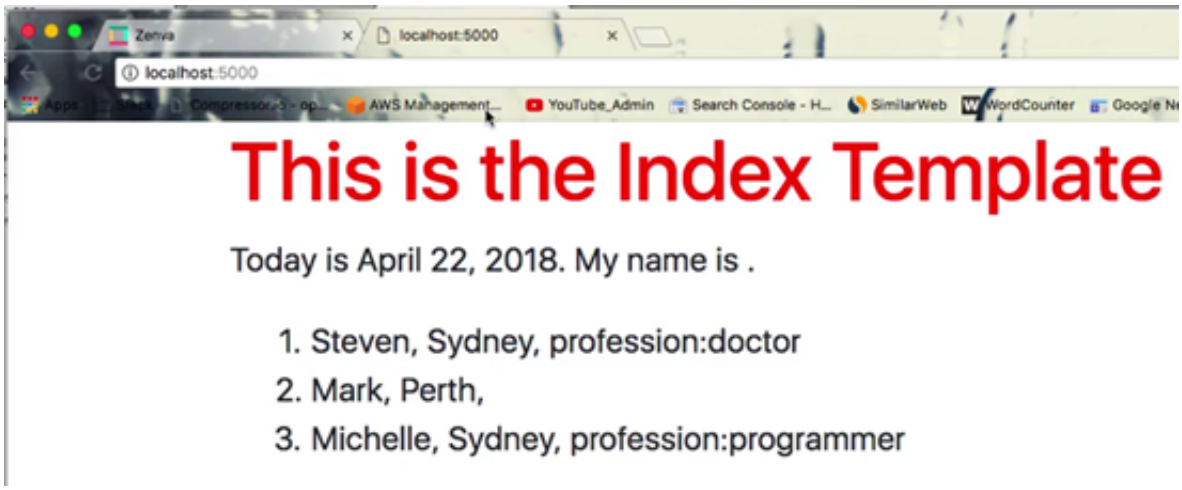
Setting Up a Conditional

In **routes/index.js**, we're going to add another **key value pair** to our **array objects**. This time, though, one of our profiles won't have the pair.

```
profiles: [
 {name: 'Steven', city:'Sydney', profession:'doctor'},
 {name: 'Mark', city:'Perth'},
 {name: 'Michelle', city:'Sydney', profession:'programmer'}
]
```

While we could set our **index.hjs** to always render the profession, doing so at the moment would result in an **empty string** for "Mark". We can get around this, this, though, by setting up a **conditional**. In **index.hjs**, we can add the code below to set it up. As you might notice, the markup for this is the exact same as before when we used it as a **for loop**. Here in this context, though, it is being used as an **if/else statement**. If it finds the **profession key**, it renders the included text. If it doesn't find the **key**, though, it doesn't render anything.

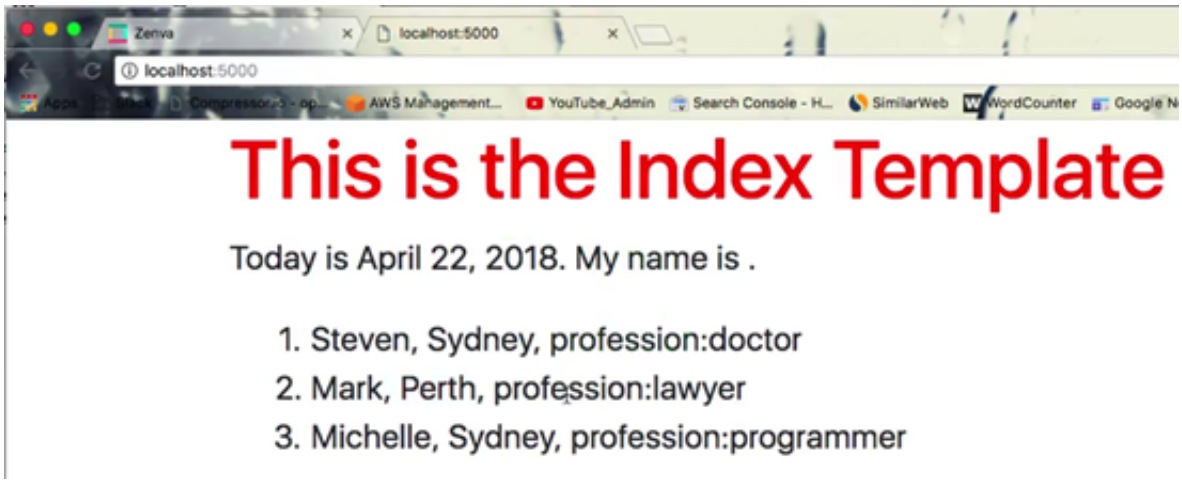
```
<ol>
  {{#profiles}}
 <li>
   {{name}}, {{city}},
   {{#profession}}
  profession:{{profession}}
   {{/profession}}
 </li>
  {{/profiles}}
</ol>
```



You can even test this in action further. Add a profession to "Mark" in **routes/index.js**.

```
{name: 'Mark', city:'Perth', profession:'lawyer'},
```

Test your app once again, and you can see that now "Mark" is rendered with his profession.



As you can see, by just using **loops** and **conditionals**, we can render data in a dynamic way without the need to hardcode individual pages. In the next lesson, we will work with handling **post forms**.

Make sure to consult the **Hogan GitHub documentation** to learn more about the **templating engine**: https://github.com/twitter/hogan.js

The code in the video for this particular lesson has been updated, please see the lesson notes below for the corrected code.

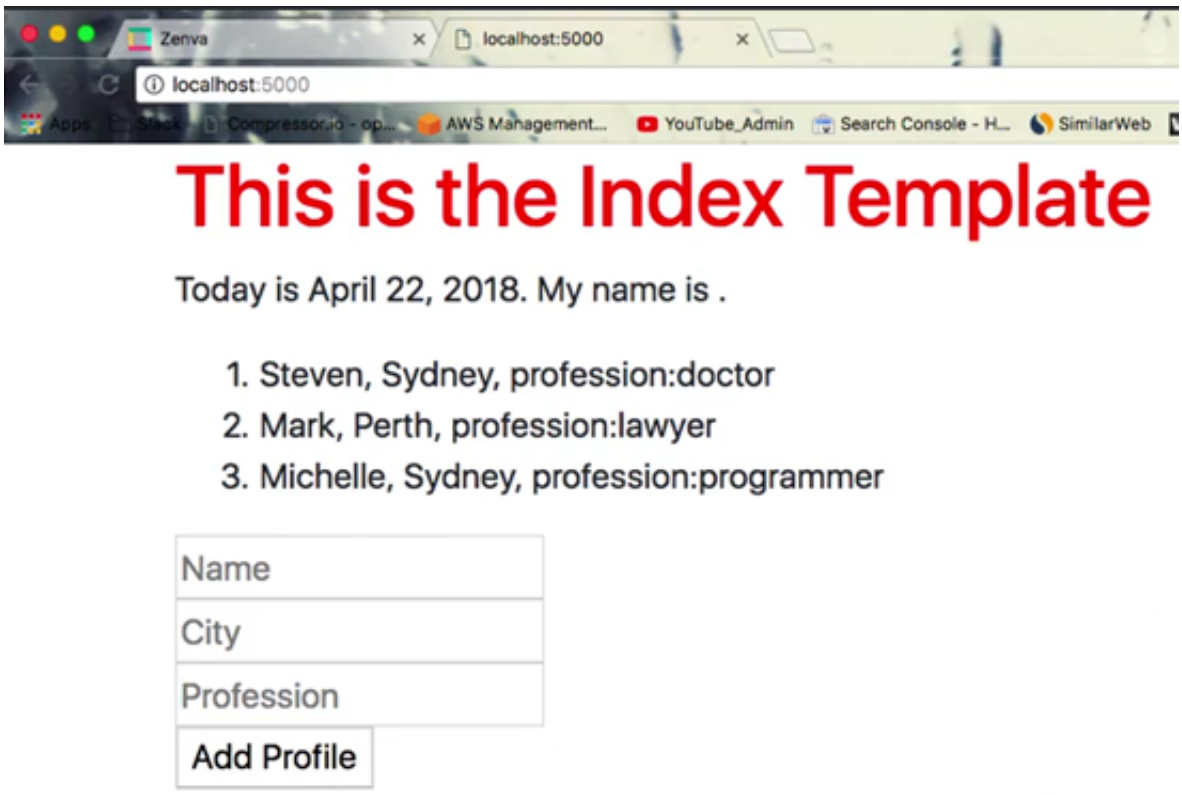
In this lesson, we're going to set up a **post request handler**. This is a common **request handler** used by web elements like forms and commenting systems. For our purposes, we're going to take in data from a form and do something with that data.

Form HTML

Within **Sublime**, open up **index.hjs**. We're going to add a form underneath our **ordered list** from the previous lesson. This form will allow the user to add more profiles to our list. As such, we need to have three **text inputs**: one for name, one for city, and one for profession. Keep in mind the **name attribute** will be used to identify the variable the data is sent as, while the **placeholder** is the actual placeholder text in the field. We will also add a **submit input** so that the profiles can be submitted.

The following code has been updated, and differs from the video:

```
<form method="post" action="/join">
 <input type="text" name="name" placeholder="Name" /><br />
 <input type="text" name="city" placeholder="City" /><br />
 <input type="text" name="profession" placeholder="Profession" /><br />
 <input type="submit" value="Add Profile" />
</form>
```



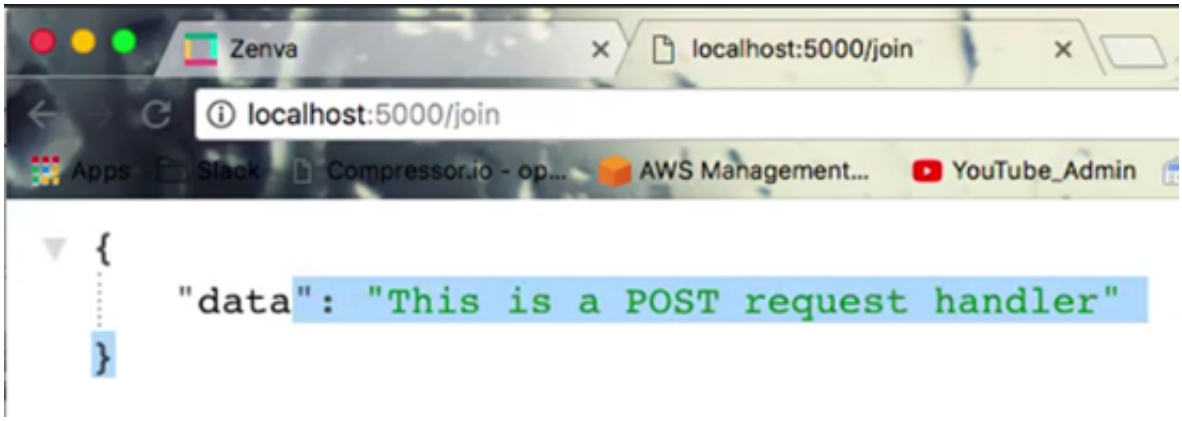
In the code above, notice the special attributes given to the **form tag**. These are *essential* to make the form work as we intend. The first one, **method**, defines the type of **request**, which for forms is almost always **post**. The second one, **action**, is the path our form will be pointed at when it's sent.

Setting up the Post Request Handler

Now that we've added the form to the page, we can work on the actual **post request handler**. Head over to **routes/index.js** and create space for the **post request handler** underneath the first **get request**. The **post handler** is formatted exactly the same as a **get request handler**, though instead of **.get** we will use **.post**. We also need to make sure to use the **path** that we defined as the **form's action** earlier. Before we do anything with profiles, let's first test this **handler** by returning some simple **JSON** with the **response**.

```
router.post('/join', (req, res, next) => {
 res.json({
        data: 'This is a POST request handler'
    })
})
```

As usual, you can use **nodemon** to get your server running. Fill out the form and then hit the button from **localhost:5000**. You should see that our **response** is what we defined in the **post handler**.



Retrieving the Data to Send

Now that we know the handler works, we need to be able to get the data in the form. To do this, we need to **import** a special **library**. In **Terminal/Command Prompt**, verify that you're in the **my-first-app directory**. Then, run the following command to **install body parser**.

```
npm install body-parser --save
```

Of course, we need to configure the project to use **body parser**. In the **main directory index.js** file, we will **import** the **library** underneath our other **imports** at the top.

```
const bodyParser = require('body-parser')
```

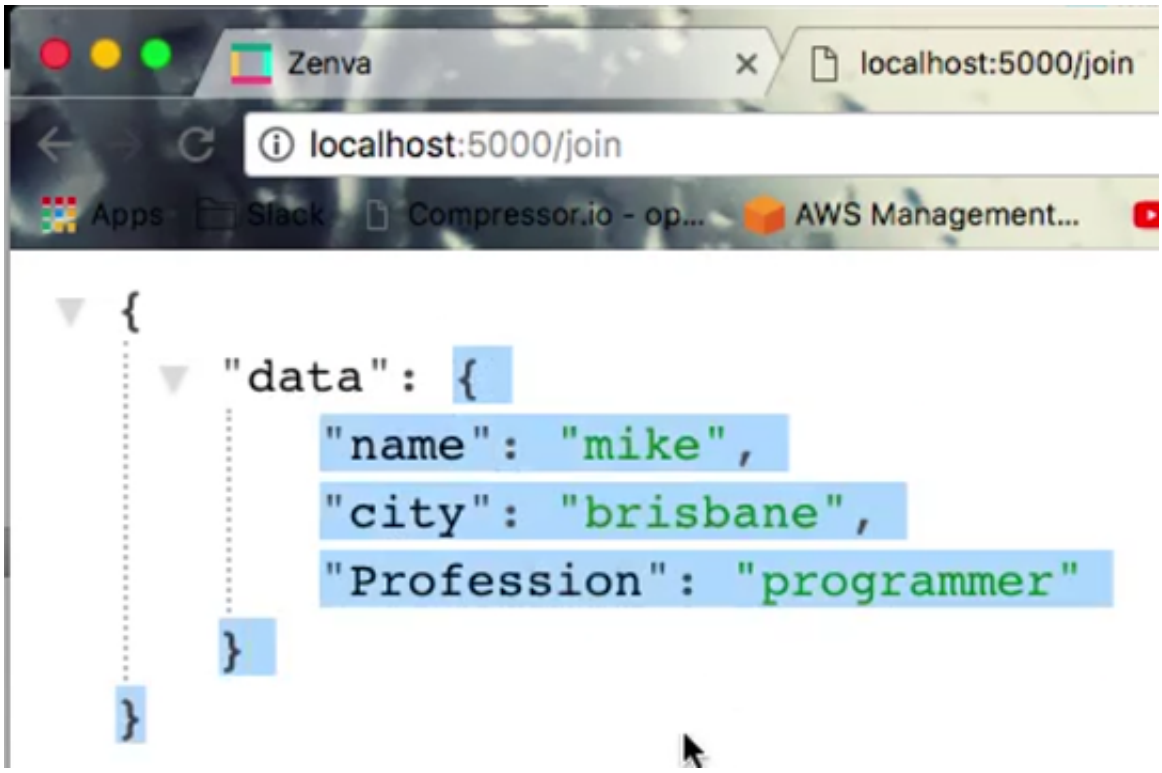
Further down the page, we will also tell the **app** to **use** the **library**. With **.urlencoded**, we are given the ability to be able to **extract** the **data** from our form when the **request** is sent.

```
app.use(bodyParser.urlencoded({extended:false}))
```


With the library in place, we can head back to **routes/index.js**. For now, we will use the **body function** we received with the **library** to parse the data out. Then, we will return the **data** as a simple **JSON object**.

```
router.post('/join', (req, res, next) => {
 const body = req.body

    res.json({
        data: body
    })
})
```

Through our test, we now see we get a profile sent back.



Pushing to our Array

The last thing we want to do is to be able to **push** the form data to our **profiles**. To begin, we're going to move our **profiles array** to be outside of the home page **get request handler**. This way, everything is more organized and we can **push** our form data to the **array**. For the **get request handler**, we just need to change it so the **profiles** key returns the **array**.

```
const profiles = [
 {name: 'Steven', city:'Sydney', profession:'doctor'},
 {name: 'Mark', city:'Perth', profession:'lawyer'},
 {name: 'Michelle', city:'Sydney', profession:'programmer'}
]
```

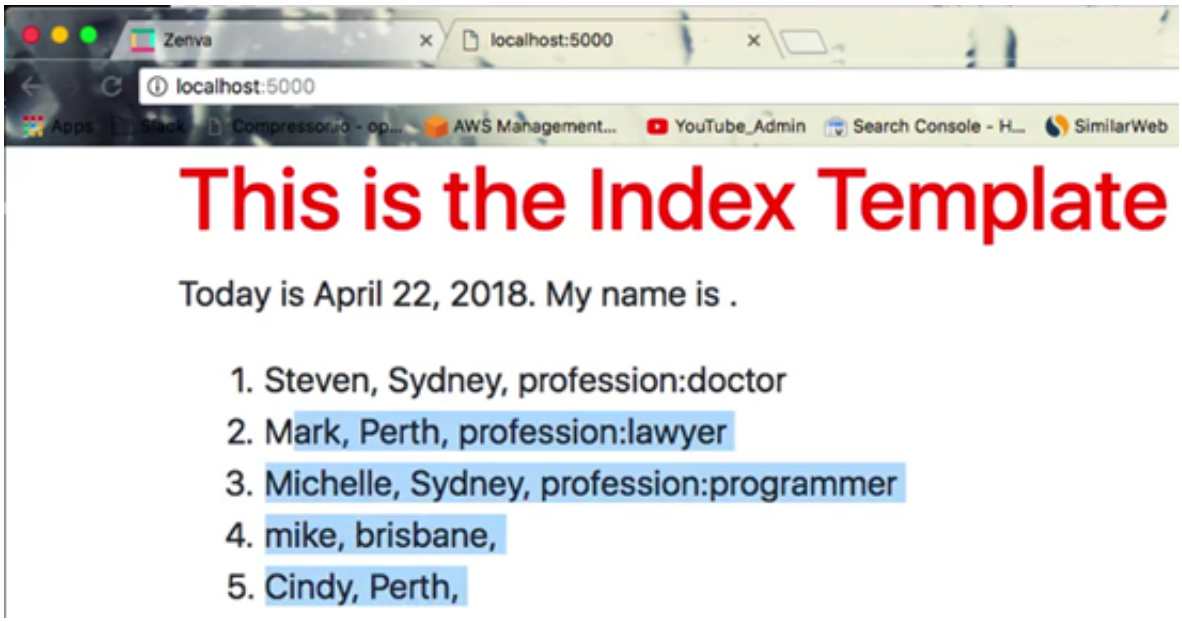


```
router.get('/', (req, res, next) => {
 const data = {
  name: 'Index',
  date: 'April 22, 2018',
  profiles: profiles
 }

 res.render('index', data)
})
```

In the **post form handler**, we can now add a line that will **push** the **data** from the form into the **array**. Additionally, we don't want to return a **JSON object** now. Thus, we will have our **response redirect** the **request** to the home page ***after*** the **array** is updated.

```
router.post('/join', (req, res, next) => {
 const body = req.body
 profiles.push(body)
 res.redirect('/')
})
```

We can test out our form now and see that it works!



In the next lesson, we'll work with using **middleware** in the **Express framework**.

You can learn more about **post requests** via the **Express documentation**:
https://expressjs.com/en/starter/basic-routing.html

In this lesson, we're going to learn about a unique element to the **Express framework** called **middleware**.

What is Middleware?

Middleware are pieces of functionality that run before **requests** to help make the **request handling** more efficient. For example, if one wanted to check before every **request** that a user was logged in, a piece of **middleware** could apply this functionality to **all requests**. For this lesson, we're going to create a simple piece of **middleware.**

However, in the **main directory's index.js** file, note that we've actually already been using **middleware**.

```
app.use(express.static(path.join(__dirname, 'public')))
app.use(bodyParser.urlencoded({extended:false}))
```

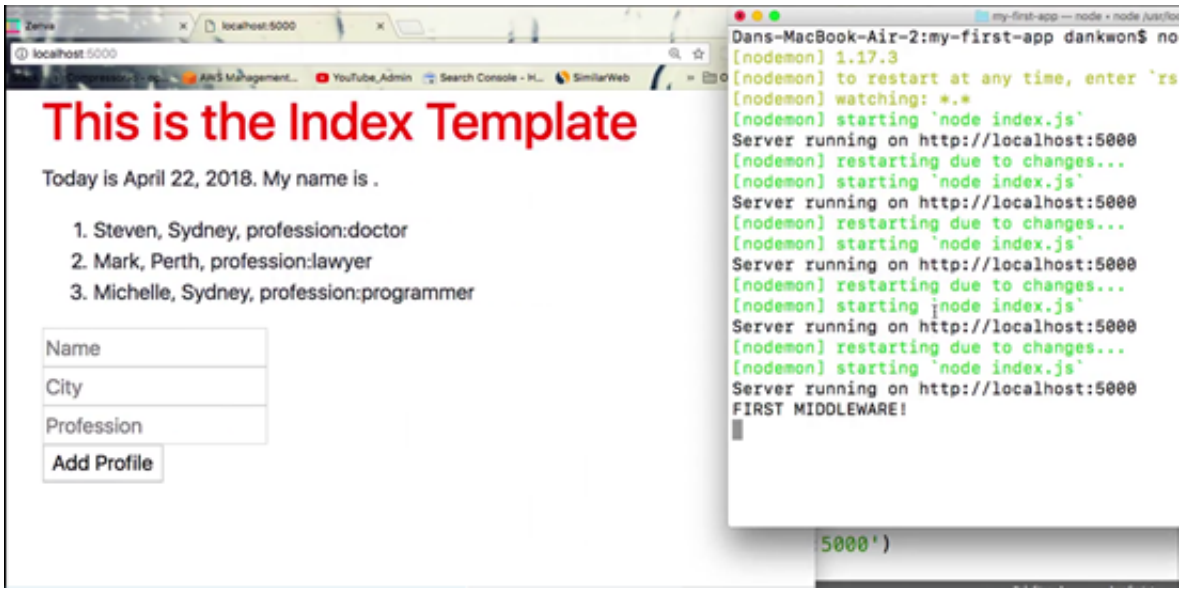
With the two lines above, we've been implementing **middleware** to use standard **functions** before our other **requests** ever occur. Thus, you can see how **middleware** is a common feature of using the **Express framework**.

Adding our Tiny Middleware

In the **main directory's index.js** file, we will now create our "first" piece of **middleware**. In order to create **middleware**, we need to use the **.use** function and, like handler's, take in a **request**, **response**, and **next** argument for a **function**. For our first test, we will simply **log** to our **console** before each and every **request**. However, we need to use the **next function** this time. By doing such, we can tell our code to move on to the **next handler** in the code once our **middleware** is done with its functionality.

```
app.use((req, res, next) => {
 console.log('FIRST MIDDLEWARE!')
    next()
})
```

Use **Terminal/Command Prompt** to run **nodemon** again and check **localhost:5000**. If you reload the page, you should see our console log appear!



Altering Request Objects

Another thing we can do with **middleware** is alter our **request object**. In the example below, we'll add a **timestamp** to our **request**. Testing the site won't show anything yet, but we can access this **timestamp variable** in subsequent **routes**.

```
app.use((req, res, next) => {
    console.log('FIRST MIDDLEWARE!')
 req.timestamp = 'April 20, 2018'
 next()
})
```

Open up **routes/index.js** and locate the **get request handler** for the home page. At the top of the **function**, we will add a **console log** that uses the **timestamp**.

```
console.log('Timestamp: ' + req.timestamp)
```

Now if you test the site and **reload** the **home page**, you will see the **timestamp** appear in the **console**.

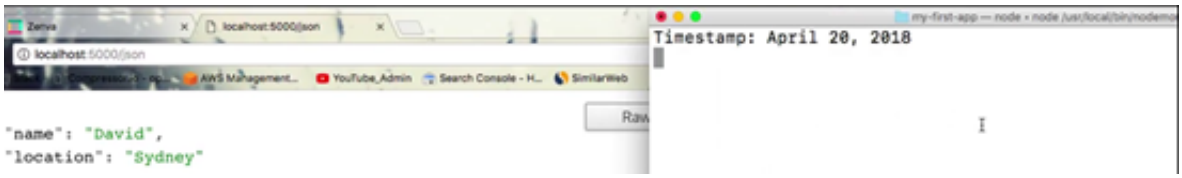
Remember that this **middleware** can function for all **routes**. Let's add this same **timestamp console log** from before to the **get request** for the **/json path**.

```
router.get('/json', (req, res, next) => {
 console.log('Timestamp: ' + req.timestamp)
 const data = {name:'David', location:'Sydney'}
 res.json(data)
```



```
})
```

If you load **localhost:5000/json**, you will see the same **console log**.



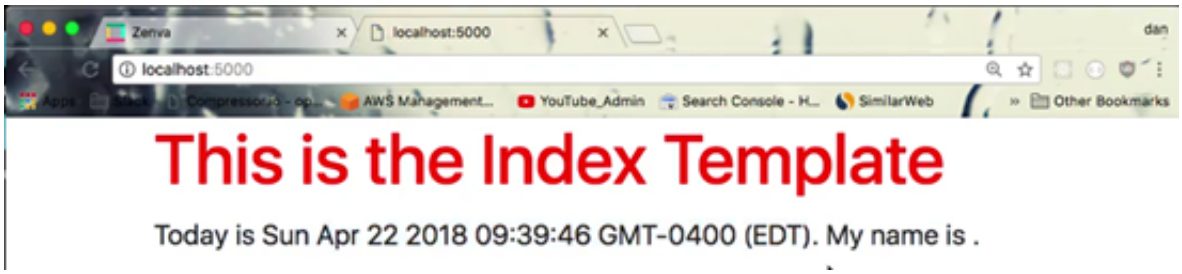
We can also do other things besides **console logs**. In the **get request handler** for the **home page**, let's change the date so it uses the **timestamp**. We'll also **remove** the **console log**.

```
router.get('/', (req, res, next) => {
 const data = {
  name: 'Index',
  date: req.timestamp,
  profiles: profiles
 }

 res.render('index', data)
})
```

Back in the **main directory's index.js** file, we will also change our **timestamp** to automatically retrieve the **date** with the **date function**.

```
app.use((req, res, next) => {
 req.timestamp = new Date().toString()
 next()
})
```

Test **localhost:5000** again, and you will see the date automatically printed for you.

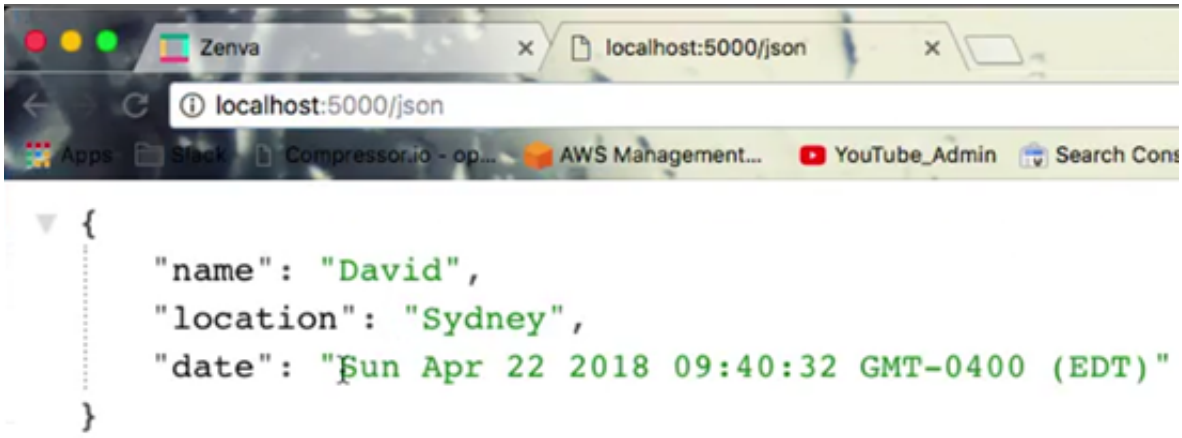


Again, remember, this **timestamp** is available on all **routes**. We will adjust the **/json path get handler** again in **routes/index.js** to get the **timestamp** there too. You will see that as expected, we still get the data automatically printed for us.

```
router.get('/json', (req, res, next) => {
 console.log('Timestamp: ' + req.timestamp)
 const data = {name:'David', location:'Sydney', date:req.timestamp}
```



```
 res.json(data)
})
```



In the next lesson, we will work on creating a simulated **login** flow for our application.

If you'd like to learn more about **middleware** in the **Express framework**, please check out the two pieces of **documentation** below:

- https://expressjs.com/en/guide/writing-middleware.html
- https://expressjs.com/en/guide/using-middleware.html

In this lesson, we're going to simulate a **login** flow that will mimic how a realistic **login system** might work.

Creating our Login Template

The first thing was want to do is create a **template** for our **login** in the **views directory**. With **Terminal/Command Prompt**, **change directories** to the **views directory** (from the **my-first-app directory**), and create a **new file**. Remember to use the command appropriate to your operating system.

```
cd views
```

Mac:

```
touch login.hjs
```

Windows:

```
echo login > login.hjs
```

With the file created, we'll now open our new **login.hjs** file in **Sublime** to create our template. We'll start first with our basic **HTML** set up. We also want to make sure to **import** our **CSS** style pages in the **head tags** and add a **div container** to our **body**.

```
<html>
<head>
 <link rel="stylesheet" type="text/css" href="/css/bootstrap.min.css">
 <link rel="stylesheet" type="text/css" href="/css/style.css">
</head>

<body>
 <div class="container">

 </div>
</body>
</html>
```

Within the **div container**, we're going to set up a login **headline** as well as a **form**. This **form** will contain a **text input** for a username and a **password input** for the password. We'll also include a **submit** button for logging in.

```
<h1>Login</h1>
<form>
 <input type="text" name="username" placeholder="Username" /><br />
 <input type="password" name="password" placeholder="Password" /><br />
 <input type="submit" value="Log In">
</form>
```

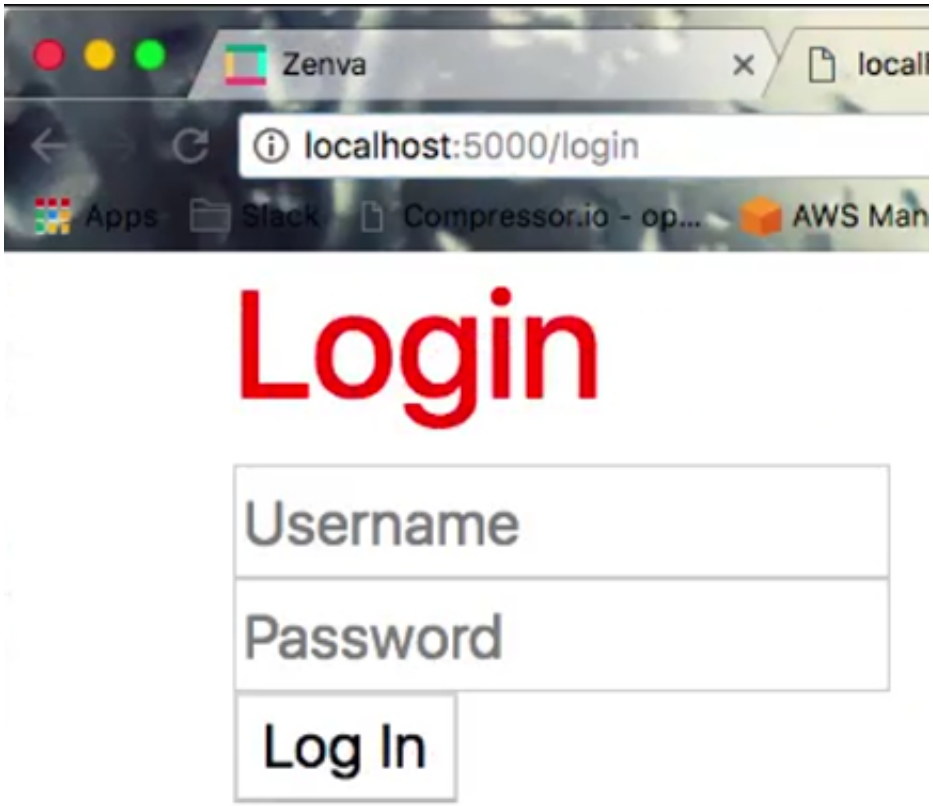

(**Note:** The password type is what allows our password to appear concealed with dots).

Request Handlers

Next, we need to head over to **routes/index.js** and set up the **path** for our login. The first thing we need is a **get request handler** that will be able to **render** our new template that we just created. We will make **/login** the path.

```
router.get('/login', (req, res, next) => {
 res.render('login', null)
})
```

At this point, you can feel free to run **nodemon** and check **localhost:5000/login** to check out the page.



We need to create a **post handler** for our **form** now, so we'll head back into **login.hjs**. As mentioned in the lesson on **post form handling**, we need to add two crucial aspects to our **form tag**: the **post method** and the **/login path action**.

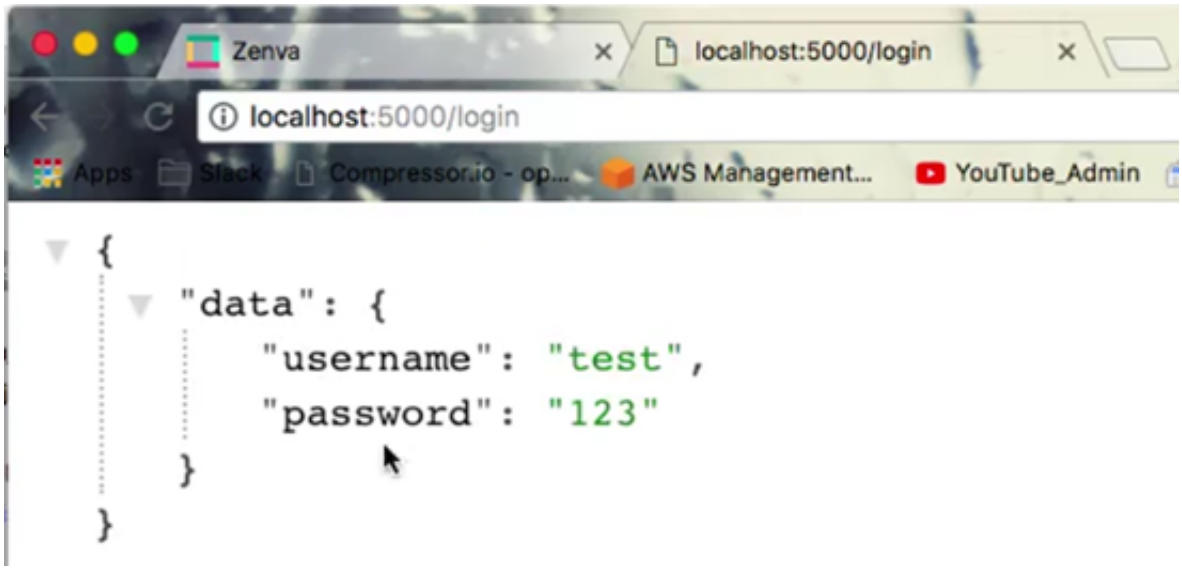
```
<form method="post" action="/login">
```

Again, back in **routes/index.js**, we'll now add in our **post request handler**. Before we do anything advanced, though, we'll simply have our handler return the **body/data** from the form to verify our

post handler works.

```
router.post('/login', (req, res, next) => {
 res.json({
  data: req.body
 })
})
```

Now you can test the **login form** and verify it's able to return the data.



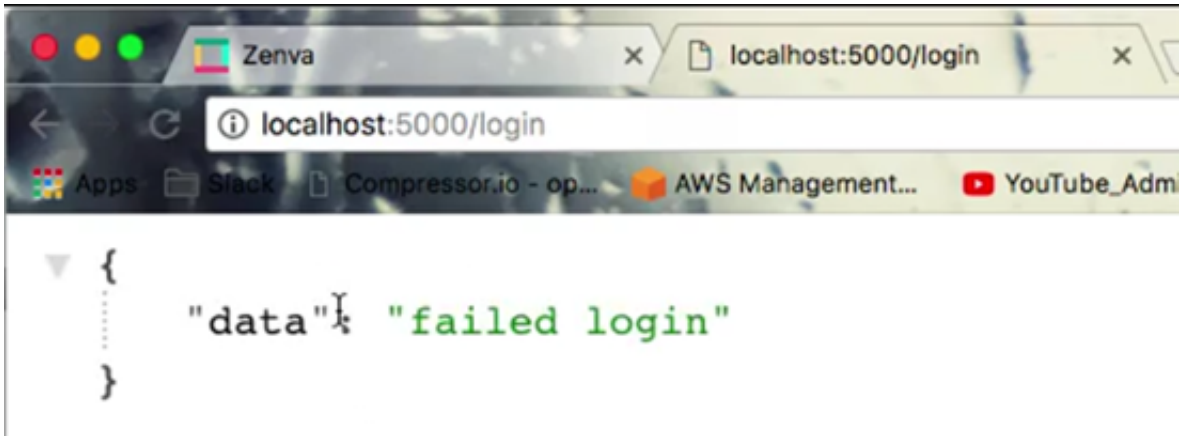
Password Verification and Showing Logged in User

With that confirmed, we will now set up a simple **if statement** to simulate password verification. For this, we're simply going to check if the password is "123". If it is, we will **redirect** to the home page. If not, we'll **respond** that the login failed.

To do this, we first need to extract the username and password into separate **variables** within our **post request handler**. After which, we can then run an **if conditional** on our password and perform the **redirect**. Note that we need to **return** out of this part of the **function** as well so the rest of the code doesn't run. For our **login failure**, we'll simply respond with a **Javascript object**.

```
router.post('/login', (req, res, next) => {
 const username = req.body.username
 const password = req.body.password

 if (password === '123'){
  res.redirect('/')
  return
 }

 res.json({
  data: 'failed login'
 })
})
```

You can once again test and verify whether the 123 password works. If it works, you should return to the home page. If not, you should see the login failure message.



However, a true login system would be able to show a user is logged in with a successful password verification. As such, let's simulate this next. Under the **imports** on **routes/index.js**, we're going to add a **variable** for a **user** set to null.

```
let user = null
```

Then, in our **post request handler** for the **login form**, we will change this variable to be the extracted username from the form when the login is successful.

```
if (password === '123'){
 user = {username: username}
 res.redirect('/')
 return
}
```

Next, in the **get request handler** for the **home page**, we'll add the **user** into the **response's data**.

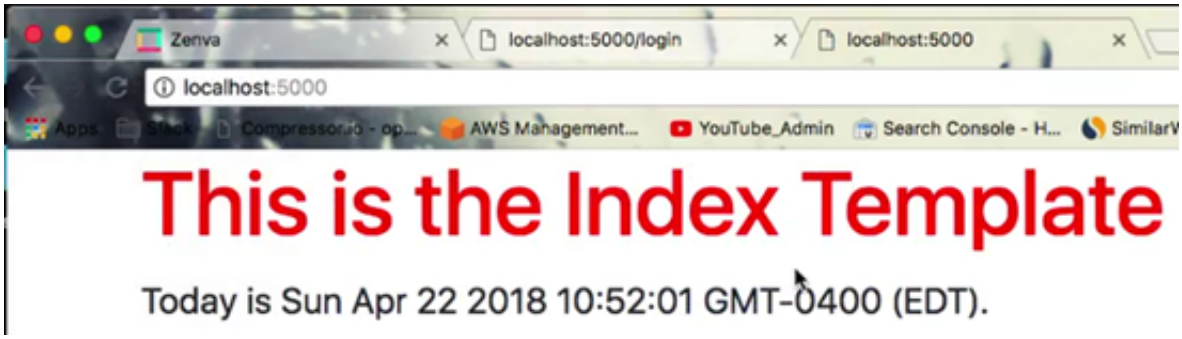
```
router.get('/', (req, res, next) => {
 const data = {
  name: 'Index',
  date: req.timestamp,
  profiles: profiles,
  user: user
 }

 res.render('index', data)
})
```

Last, we'll open up **index.hjs** and alter the home page's paragraphs. We want to change our paragraph so that **if** the **user** is **not null** (i.e. the user is logged in), we will print a message with their **username** from the **user variable**. Like we did with our **list**, we can do this easily with our

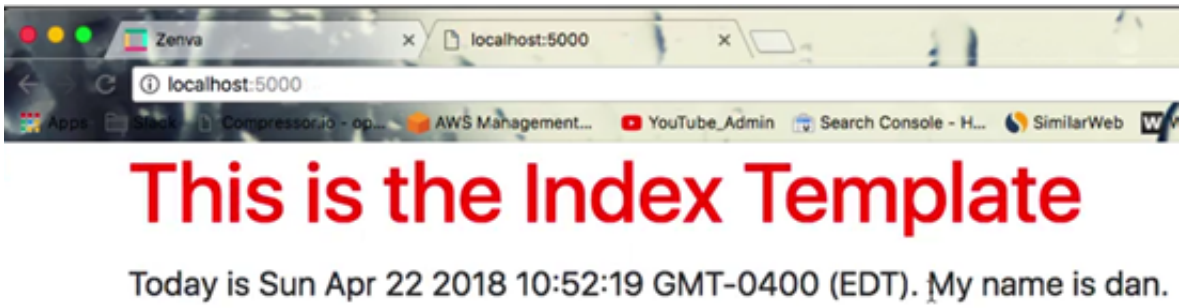
simple **conditional** markup.

```
<p>
 Today is {{date}}.
 {{#user}}
 My name is {{user.username}}.
 {{/user}}
</p>
```

Let's test our login now. When we go to the home page at **localhost:5000**, we should only receive the data message in the paragraph.



Once we "login", though, we should see our **conditional statement printed**.



In the next and final lesson, we're going to render a better login failure page.

In this final lesson, we're going to set up a page render for when our login fails from the last lesson.

Error Page Setup

From **Terminal/Command Prompt**, we're going to move into the **views directory** from our main **my-first-app directory**. Then, we'll create a **new file** that we can use as a **template** for our log in failure.

```
cd views
```

Mac:

```
touch error.hjs
```

Windows:

```
echo error > error.hjs
```

In **Sublime**, we can now open up our new **error.hjs** page and create its **HTML**. Like before, we'll start with our skeleton structure, our **CSS links** in the **head tags**, and a **div container** in the **body**. Within the **div container**, we will set up a simple **headline** and a paragraph for a **message**.

```
<html>
<head>
 <link rel="stylesheet" type="text/css" href="/css/bootstrap.min.css">
 <link rel="stylesheet" type="text/css" href="/css/style.css">
</head>

<body>
 <div class="container">
  <h1>Error</h1>
  <p>{{message}}</p>
 </div>
</body>
</html>
```

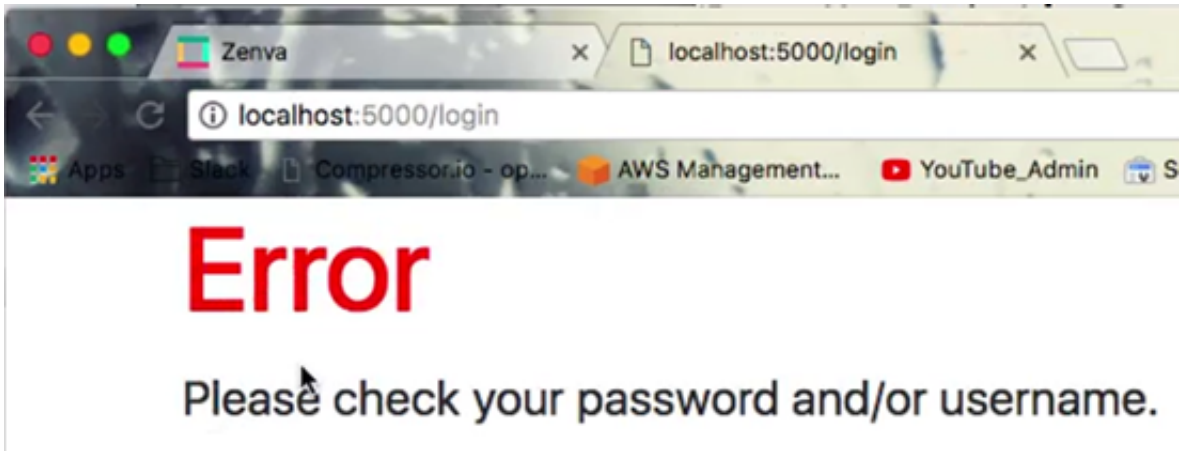
Now we can head over to **routes/index.js** and change the **post request handler** for our login. Instead of rendering **JSON** when the login fails, we want to **render** our **error.hjs template** with the **.render function**. Further, since we can pass in **data** with our our **template**, we'll create a simple **message** that we can send.

```
router.post('/login', (req, res, next) => {
 const username = req.body.username
 const password = req.body.password

 if (password === '123'){
  user = {username: username}
```



```
  res.redirect('/')
  return
 }

 const data = {
  message: 'Please check your password and/or username.'
 }

 res.render('error', data)
})
```

From the **my-first-app directory**, we can now run **nodemon** and test it out. Enter the wrong password, and you should see our new **error** page rendered.



Since our **error page** is generic, we could use this for any number of error producing scenarios. All we would have to do is pass in a different **message** in any **handler** to indicate what sort of **error** occurred.

Congratulations! You've now completed **Zenva's Express for Beginners** course. While this was just a cursory overview of the **Express framework**, this should be a good first step to developing your own web applications. If you'd like more information about the **Express framework**, don't hesitate to check out the **Express framework documentation**: https://expressjs.com/